Teaching Graphics with the OpenGL Shading Language

Jerry O. Talton
Departments of Computer Science
University of California, Santa Cruz
Stanford University
jtalton@cs.stanford.edu

Darren Fitzpatrick
Department of Computer Science
University of California, Santa Cruz
darrenf@soe.ucsc.edu

ABSTRACT

A long-standing difficulty in the development of introductory courses in computer graphics is balancing the educational necessity of ensuring mastery of fundamental graphical concepts with the highly desirable goal of exciting and inspiring students to further study by enabling them to produce visually interesting programming projects. Recently, we have developed a modified curriculum predicated on the extensive integration of the OpenGL Shading Language with a more traditional pedagogical approach. We utilized this curriculum in the quarter-long, upper-division introductory graphics course taught in the Department of Computer Science at the University of California, Santa Cruz. Our experience indicates that making shading an integral part of the entry-level curriculum inculcates students with a comprehensive understanding of the algorithms and mathematical concepts that underlie modern graphical systems, while simultaneously equipping them with the tools necessary to produce complex projects with state-of-the-art technology.

Categories and Subject Descriptors: K.3.2 [Computers and Education]: Computer and Information Science Education; I.3.0 [Computer Graphics]: General

General Terms: Design, Languages, Human Factors

Keywords: OpenGL shading language, graphics, pedagogy

1 INTRODUCTION

The problem of developing an introductory course in computer graphics that is both sufficiently broad and deep is well established in the field of computer science education. While it is generally agreed that interactive, realtime curricula are more suitable for new graphics students than their offline, rendering-oriented counterparts [6, 8], determining which specific material is of maximum import, as well as the precise manner in which to present this material, remain subjects of continuing debate [2].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee, provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee. So there.
SIGCSE'07, March 7–10, 2007, Covington, Kentucky, USA.
Copyright 2007 ACM 1-59593-361-1/07/0003 . . . $5.00.

Existing pedagogical approaches may be broadly categorized as either *top-down*, in which students use one of many popular graphics APIs (such as OpenGL) to incrementally build complex graphical applications [14], or *bottom-up*, in which students develop toy implementations of each stage of the graphics pipeline [12]. In practice, neither of these methods has proven wholly satisfactory. In the top-down approach, students are at risk of becoming embroiled in the technical idiosyncrasies of their chosen API, often to the marked detriment of their theoretical understanding of the underlying material. Conversely, in the bottom-up approach, the sheer number of special cases that must be handled in order to develop a robust and useful implementation of the graphics pipeline may prevent students from ever producing non-trivial graphical systems. Since many students are drawn to graphics by a desire to emulate the visual effects they see in video games and movies, this latter outcome can be especially egregious, particularly given the widely recognized potential of graphics courses to excite students and motivate them to further study [8].

Hybrid approaches have been implemented with varying degrees of success [5], but the fundamental difficulty of ensuring that students fully understand the algorithms and mathematical concepts that underlie modern graphical systems, while simultaneously enabling them to produce visually interesting projects, remains largely intractable in many introductory graphics courses. Recent consensus in the educational graphics community has advocated the removal of raster-level algorithms from introductory curricula, while also acknowledging that students wishing to enter the graphics industry or pursue graduate study in the field may be significantly disadvantaged by this approach [2]. In most semester- or quarter-long courses, however, there is simply not enough time for students with no previous graphical experience to implement low-level algorithms such as clipping or rasterization and also produce a complex, API-driven project.

Another significant hurdle that must be overcome anew with each successive course offering is that the rate of technological progress in the field of graphics far outstrips that of other, more firmly established areas. For roughly the past ten years, graphics hardware has improved at a rate nearly three times faster than Moore's Law predicts, effectively doubling in processing power every six months [9]. As a consequence of this fact, the advanced material presented in a semester-long graphics course may, in theory, become obsolete between the time it is placed on the syllabus and the time it is presented in class. Clearly every effort should be made to keep courses as relevant and up-to-date as possible, but this is a considerable challenge in graphics classes given the rapid pace of technological advancement.

Recently, in an effort to combat these problems in the Department of Computer Science at the University of California, Santa Cruz, we have developed a modified curriculum predicated on the extensive integration of the OpenGL Shading Language with a more traditional top-down pedagogical approach. We utilized this curriculum in the quarter-long, introductory graphics class offered by the department: an upper-division course primarily intended for upperclassmen and first-year graduate students with no previous graphical exposure. Our experiences indicate that making shading an integral part of the entry-level curriculum engenders a wide variety of educational benefits. In particular, we believe that teaching shaders bridges the gap between top-down and bottom-up approaches by exposing the most relevant low-level technical details of the graphics pipeline in a manner that directly contributes to the production of complex, high-level software. We submit that students who learn the OpenGL Shading Language as part of their introduction to computer graphics necessarily gain a better grasp of the mathematical underpinnings of modern graphical algorithms and are more likely to produce visually and technologically impressive course projects than those who are taught to use only the fixed-function OpenGL API.

2 OPENGL SHADING LANGUAGE

The OpenGL Shading Language (GLSL) is a high-level procedural programming language that allows application programmers to control the graphics units that are used to process vertex and fragment data in the OpenGL pipeline. The language was created by the OpenGL Architecture Review Board and is part of the OpenGL 2.0 standard. GLSL has been supported (to varying degrees) on virtually all consumer graphics hardware produced since 2002.

Since GLSL is based on C and C++ syntax and flow control, it is extremely accessible to students already familiar with either language. In addition, the seminal reference text on the subject [11] contains a near-complete technical specification as well as thousands of lines of example code, making it virtually ideal for the classroom setting. One particularly attractive feature of GLSL is that it is very similar to the other popular shading languages used in graphics, such as NVIDIA's Cg, Microsoft's High Level Shading Language (HLSL), and even the production-oriented RenderMan Shading Language (RSL). As a result, a working knowledge of shaders in OpenGL translates directly to many other platforms and contexts.

Traditionally, educational use of GLSL has been relegated to advanced courses on realtime rendering, despite the fact that shaders are ubiquitous in modern graphical applications. Non-realtime rendering has relied exclusively on shaders to describe lighting and material properties for years, and virtually all modern interactive graphical software uses the programmable pipeline in some manner. In fact, the vertex and pixel stages of most current-generation graphics hardware are entirely programmable, and rely on a set of default shaders to emulate the fixed-function pipeline when necessary.

```
varying vec3 N;

void main()
{
  gl_Position = gl_ProjectionMatrix *
                gl_ModelViewMatrix * gl_Vertex;
  N = gl_ModelViewMatrixInverseTranspose * gl_Normal;
  N = normalize(N);
}
```

Figure 1. A simple GLSL vertex shader.

3 EDUCATIONAL BENEFITS

Teaching shading in an introductory setting has many benefits over the traditional approach of focusing exclusively on the fixed-function OpenGL API. Perhaps the most compelling argument for teaching GLSL is that writing shaders forces students to understand the mathematical transformations that lie at the root of the graphics pipeline. This is because the OpenGL Shading Language shifts the responsibility of applying transformations from the application programming interface to the application programmer. In Figure 1, a simple GLSL vertex shader—roughly the OpenGL Shading Language equivalent of "Hello World"—is shown. The shader transforms a vertex position from object-space to clip-space and computes the associated eye-space vertex normal: two tasks that are traditionally handled transparently by the OpenGL API. In order for a student to write this shader, he or she must explicitly understand the object-space, eye-space, and clip-space coordinate systems, as well as the non-commutativity of matrix multiplication, and the inverse transpose transformation for normal vectors. Since GLSL syntax is extremely straightforward, students have a direct connection with the mathematical equations governing their applications, and the precise behavior of each stage of the pipeline is no longer obscured by esoteric combinations of API settings and options.

This unmasking is even more profound in more complicated algorithms. Figure 2 shows the C++ code necessary (after texture setup) to perform cubic environment mapping using the fixed-function OpenGL API. While this code segment seems to indicate that some feature involving a combination of textures, cubes, and cartography has been enabled, it fails to provide any insight at all as to what cube mapping is or how it works. In stark comparison, the equivalent GLSL code, which is displayed in Figure 3, is quite elucidating. A student writing this shader must calculate the reflected eye vector, perform the corresponding texture lookup, and set the output color of the fragment. This example, while somewhat contrived, illustrates one of the fundamental advantages of teaching GLSL: shaders expose functionality the traditional API obscures.

Two other areas in which GLSL affords students virtually unparalleled opportunities for learning are lighting and

```
glEnable(GL_TEXTURE_CUBE_MAP);
```

Figure 2. Cube Mapping using the fixed-function OpenGL API.

```
varying vec3 N;
varying vec3 E;

uniform samplerCube envMap;

void main()
{
   vec3 R = 2 * dot(N, E) * N - E;
   vec3 color = vec3(textureCube(envMap, R));
   gl_FragColor = vec4(color, 1.0);
}
```

Figure 3. Cube Mapping in a GLSL fragment shader.

shading. The fixed-function OpenGL pipeline has no support for per-pixel shading (instead relying on the antiquated and tessellation-dependent Gouraud method) and provides only a single lighting model (Blinn-Phong). Using the OpenGL Shading Language, however, students can use fragment shaders to directly implement arbitrarily complex lighting models and see firsthand the effect of changing shading parameters: a process that has been largely impossible in traditional top-down realtime curricula. This enhanced functionality not only allows students to easily produce more realistic three-dimensional applications than ever before, but also gives them an intuitive, practical understanding of the basic quantities and relationships involved in light transport. In addition, teaching GLSL largely eliminates the need to spend instructional time covering unwieldy and outdated algorithms such as texture shading.

Another benefit of teaching GLSL is that shader programming represents a relatively unique kind of computing. While most computer science education focuses on sequential programming languages and techniques, as silicon fabrication processes edge closer and closer to fundamental physical limits, modern microprocessor architectures are becoming more and more parallelized. For many students, programming graphics hardware may be their first experience with stream processing. By incorporating render-to-texture and multipass methods into the curriculum, students can begin to consider parallel algorithms as a natural extension of more familiar techniques.

4 CURRICULUM

Our curriculum loosely follows the general structure of [1]. We prefer this text for several reasons. First, it presents a top-down, OpenGL-based approach to introductory graphics, yet still provides a competent treatment of low-level algorithms like rasterization. Second, it has sufficient scope to ensure that most topics we discuss in class are also covered in the text. Third, its presentation is highly mathematical in nature, and excellent, detailed derivations are provided where appropriate. Lastly, it is one of the few introductory graphics textbooks that has adequate coverage of basic programmable shaders.[1] We supplement our primary text with additional readings from [13, 11].

[1]It should be noted, however, that the first printing of the fourth edition contains several conspicuous errors in the chapter on programmable shading. Students should be made aware of this problem early on and directed to the author's errata on the web.

Somewhat surprisingly, integrating GLSL into an existing top-down curriculum is not a monumentally difficult task. We spend only two full lectures explicitly covering the programmable pipeline: the first towards the beginning of the quarter, focusing on the OpenGL Shading Language syntax and setup, and the second in the final weeks of the class, discussing advanced rendering techniques and algorithms. We supplement other lecture topics with discussion of shader-level implementation concepts, but this usually consists of little more than ensuring that we give complete and correct mathematical descriptions of the relevant graphical phenomena.

It is important to understand that we find there to be little value in teaching GLSL for its own sake in an introductory graphics setting. Rather, we explore a broad and diverse curriculum and use the OpenGL shading language as a tool to extend and enhance student understanding. Indeed, many traditional graphics topics besides transformations and lighting can benefit from judicious application of the programmable pipeline. Vertex shaders, for instance, provide a natural platform upon which to implement efficient, physically-based particle simulations. Image processing algorithms, when firmly rooted in the theoretical foundations of separable convolutions, lend themselves directly to multi-pass render-to-texture approaches. Simple raytracing can even be performed in a fragment shader to approximate refractive effects and compute visibility for shadowing.

In a quarter-long course, we have time for only three substantial programming assignments. The first is a short assignment that focuses on geometry processing and does not use the programmable pipeline. The second introduces students to GLSL, and requires them to implement a variety of vertex and fragment shaders. The final project for the course is open-ended and intended to represent one of the most significant undertakings of the students' college careers. To give students a starting point for their assignments, we provide complete source code for a simple two-dimensional single-polygon animation in both fixed-function and GLSL form.

One particularly nice feature of the OpenGL Shading Language is that shader files are compiled dynamically at runtime. In a semester-long course, we would introduce another kind of introductory GLSL assignment. Students would be provided with a pre-built application executable and a detailed interface specification, and subsequently be required to write only the GLSL shaders without having to worry about setup code at first. This strategy is similar to the approach described in [3], but requires no intermediary software layer to be introduced between shaders and application.

5 TECHNOLOGICAL CONSIDERATIONS

Although the OpenGL Shading Language is part of the OpenGL 2.0 standard, some standards appear to be more standard than others. Due to Microsoft's desire to promote its Direct3D API, OpenGL support in Windows is severely lacking. Currently available versions of Microsoft's operating systems provide native support only for OpenGL 1.1.[2] As a result, Windows applications that utilize mod-

[2]With the upcoming release of Windows Vista, Microsoft will be advancing boldly to the year 2002 with "native" support for OpenGL 1.4 through a Direct3D emulation layer.

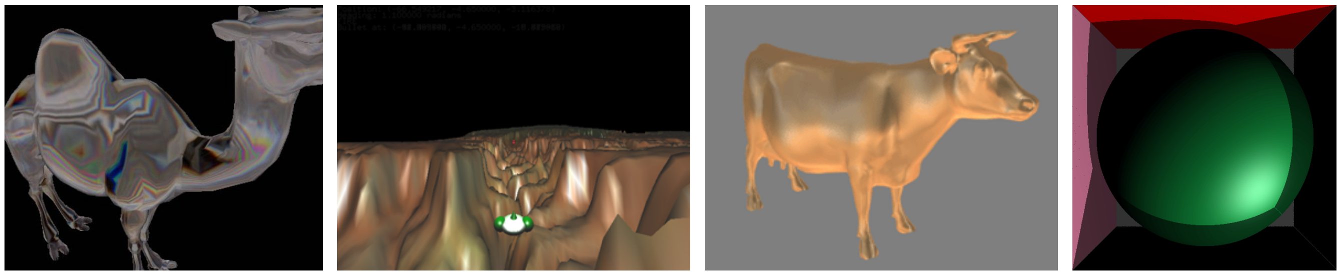

Figure 4. Screenshots of some of the final course projects that made use of GLSL. From left to right: chromatic aberration on a camel (a), a space shooter with per-pixel lighting and particle effects (b), complex illumination from dynamically-filtered environment maps (c), dynamic shadows using realtime raytracing in a fragment shader (d).

ern OpenGL features must dynamically locate the necessary function pointers by querying the installed vendor-supplied graphics drivers. Linux support for GLSL is equally problematic and vendor dependent, and the OpenGL Extension Wrangler Library [7] is a virtual necessity for making the experience bearable on either platform.

Macintosh OS X, in stark contrast, has had native GLSL support since version 10.4.3. This fact, combined with the wide range of open-source and Apple-provided development tools, makes it a more natural choice for the development platform in a GLSL-based course.

One final technological consideration is that older graphics cards may not fully support all OpenGL Shading Language features. In particular, less recent hardware may lack true branching functionality and thus incur substantial performance penalties during the evaluation of loops and conditional expressions. Similarly, some very old graphics cards may have prohibitive restrictions on the maximum number of instructions a single shader may contain, causing the driver to fall back to software emulation mode for complex shader programs. Nevertheless, these limitations are largely tolerable in an educational setting, and institutions with access to hardware from the NVIDIA FX or ATI 9700 series or later are unlikely to encounter significant problems.

6 STUDENT REACTION

With a class of only 24 students, any serious attempt at statistical analysis of student response to our curriculum would be sorely misguided. Likewise, we recognize that graphics courses are naturally predisposed to glowing student reviews irrespective of their actual merit. Nonetheless, we do believe that the anecdotal evidence offered by our students' anonymous course evaluations has some instructive value.

Overall, students were extremely enthusiastic about the course material. As Table 1 indicates, 84% of the class rated the quality of the course assignments as "Very Good" or "Excellent". In addition, in their evaluations, students described the class as:

> "By far the best class I've taken. Anywhere."

> "One of the greatest learning experiences ever."

> "The best class that I have taken in my four years at UCSC."

Not all of the feedback was positive, however, suggesting that there is still room for improvement in our approach. In particular, some students were frustrated with the Macintosh as a development platform, as well as the relative sensitivity of the GLSL setup code. One student opined:

> "The hardest part of [the second programming assignment] was setting up the environment."

One measure of student engagement is the effort that students expended in the course. Table 2 illustrates that most students rated their level of exertion as considerable. Quantitatively, 42% of students indicated that they spent 18 hours or more per week outside of class studying and completing course assignments. Students characterized their work habits as follows:

> "I put more effort into this course than I ever do."

> "We covered material that actually matters, so it was worth learning."

Students were allowed to design their own final course projects, with the sole restriction that the end result had to be sufficiently visually interesting. We provided a long and varied list of project suggestions, and students were explicitly not required to pick topics associated with programmable shading. Nonetheless, 16 of the 23 students who turned in a project picked a topic that made substantial use of the OpenGL Shading Language. A small gallery of some of the more impressive GLSL projects can be seen in Figure 4.

Poor	Fair	Satisfactory	Very Good	Excellent
0%	0%	16%	26%	58%

Table 1. Students' rating of the quality of the course's assignments.

Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
0%	0%	0%	25%	70%

Table 2. Students' response to the statement "I put considerable effort into this course."

7 CONCLUSIONS AND FUTURE WORK

In this paper we have described a method for integrating the OpenGL Shading Language into the introductory graphics curriculum. While further refinement of our approach is certainly warranted, we strongly feel that the underlying strategy was a success. Using GLSL as a major component of introductory courses in computer graphics provides students with cutting-edge tools with which to create exciting graphical applications and greatly enhances their understanding of fundamental course topics and algorithms. Furthermore, as graphics hardware continues its exponential technological advancement, we believe that programmable shaders will play an increasingly pronounced role in realtime applications, and that students who have no prior experience with shading will be at a significant disadvantage.

One area that appears particularly ripe for future inclusion in our curriculum is general purpose computing on graphics processors (GPGPU). Once students have mastered the use of programmable hardware for transformations and shading, it is relatively straightforward to demonstrate the utility of graphical architectures for solving non-graphical problems, especially given the recent advent of GPGPU systems [4] and metaprogramming environments [10]. Due to the prodigious processing power and ubiquity of modern graphics hardware, a brief introduction to the field of GPGPU could be of enormous benefit even to students whose eventual computational careers are far removed from the realm of graphics and visualization.

8 WEB RESOURCES

Our course webpage is archived online at:

`http://www.soe.ucsc.edu/classes/cmps160/Spring06/`

The course syllabus, lecture notes, assignments, and distributed sample code are all available for download.

9 ACKNOWLEDGMENTS

The authors would like to thank Keenan Crane for his enlightening guest lecture on GPU algorithms, Jesse Hall for clarifying several sticky technical matters, Michael Garland for many helpful discussions, Lynn Maxwell and Jacqui Talton for their careful proofreading, and James Davis for making this work possible. The project images in this paper are courtesy of Tim Babb, John Holsworth, Cynthia Lee, and Krystle de Mesa. Thanks also to the authors' advisors, Ron Fedkiw and Suresh Lodha, for their forbearance during the preparation of this paper. Lastly, the authors are deeply indebted to David Luebke, whose comments at NVIDIA U in 2004 were the inspiration for this work. The first author was partially supported by a Stanford University School of Engineering Fellowship.

References

[1] E. Angel. *Interactive Computer Graphics: A Top-Down Approach Using OpenGL*. Addison-Wesley, fourth edition, 2005.

[2] E. Angel, S. Cunningham, P. Shirley, and K. Sung. Teaching computer graphics without raster-level algorithms. In *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education*, pages 266–267, New York, NY, USA, 2006. ACM Press.

[3] M. Bailey. Teaching OpenGL shaders: Hands-on, interactive, and immediate feedback. In *Eurographics '06 Educational Papers*, September 2006.

[4] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. *SIGGRAPH '04: Proceedings of the 32nd annual conference on Computer graphics and interactive techniques*, 23(3):777–786, 2004.

[5] S. Cunningham. Powers of 10: the case for changing the first course in computer graphics. In *SIGCSE '00: Proceedings of the 31st SIGCSE technical symposium on Computer science education*, pages 46–49, New York, NY, USA, 2000. ACM Press.

[6] L. Hitchner, S. Cunningham, S. Grissom, and R. Wolfe. Computer graphics: the introductory course grows up. In *SIGCSE '99: The proceedings of the 30th SIGCSE technical symposium on Computer science education*, pages 341–342, New York, NY, USA, 1999. ACM Press.

[7] M. Ikits and M. Magallon. The OpenGL extension wrangler library. Available at: `http://glew.sourceforge.net`. Accessed September 7th, 2006.

[8] Joint Task Force on Computing Curricula. Computing curricula 2001. *Journal on Educational Resources in Computing*, 1(3es):1, 2001.

[9] M. Kilgard. NVIDIA graphics, Cg, and transparency. *SIGGRAPH 2006 Course Notes*, 2006.

[10] M. McCool, S. D. Toit, T. Popa, B. Chan, and K. Moule. Shader algebra. *SIGGRAPH '04: Proceedings of the 32nd annual conference on Computer graphics and interactive techniques*, 23(3):787–795, 2004.

[11] R. J. Rost. *OpenGL Shading Language*. Addison-Wesley Professional, second edition, 2006.

[12] P. Shirley. *Fundamentals of Computer Graphics*. A. K. Peters, Ltd., second edition, 2005.

[13] D. Shreiner, M. Woo, J. Neider, and T. Davis. *OpenGL Programming Guide: The Office Guide to Learning OpenGL Version 2*. Addison-Wesley Professional, fifth edition, 2005.

[14] K. Sung and P. Shirley. A top-down approach to teaching introductory computer graphics. In *SIGGRAPH '03: Educators program from the 30th annual conference on Computer graphics and interactive techniques*, pages 1–4, New York, NY, USA, 2003. ACM Press.